8.4 Analyzing Algorithm Running Time

Let us consider a very similar function to `print_integers` from the beginning of the chapter:

```
def print_items(lst: list) -> None:
    for item in lst:
        print(item)
```

Here, `print_items` takes a list as input instead, and so n is equivalent to `len(lst)`.[1] How can

[1] For the remainder of this course, we will assume input size for a list is always its length, unless something else is specified.

we use our asymptotic notation to help us analyze the running time of this algorithm? Earlier, we said that the call to `print` took 1 “basic operation”, but is that true? The answer is, it doesn’t matter. By using asymptotic notation, we no longer need to worry about the constants involved, and so don’t need to worry about whether a single call to `print` counts as one or ten “basic operations”.

Just as switching from measuring real time to counting “basic operations” allows us to ignore the computing environment in which the program runs, switching from an exact step count to asymptotic notation allows us to ignore machine- and programming language-dependent constants involved in the execution of the code. Having ignored all these external factors, our analysis will concentrate on how the **size of the input** influences the running time of a program, where we measure running time just using asymptotic notation, and not exact expressions.

Warning: the “size” of the input to a program can mean different things depending on the type of input, or even depending on the program itself. Whenever you perform a running time analysis, be sure to clearly state how you are measuring and representing input size.

Because constants don’t matter, we will use a very coarse measure of “basic operation” to make our analysis as simple as possible. For our purposes, a basic operation (or step) is **any block of code whose running time does not depend on the size of the input**.[2]

[2] To belabour the point a little, this depends on how we define input size. For integers, we usually will assume they have a fixed size in memory (e.g., 32 bits), which is why arithmetic operations take constant time. But of course if we allow numbers to grow infinitely, this is no longer true, and performing arithmetic operations will no longer take constant time.

This includes all primitive language operations like most assignment statements, arithmetic calculations, and list and string indexing. The one major statement type which does not fit in this category is a function call—the running time of such statements depends on how long that particular function takes to run. We'll revisit this in more detail later.

The runtime function

The running time of `print_items` depends *only* on the size of the input list, and not the contents of the list. That is, we expect that `print_items` takes the same amount of time on every list of length 100. We can make this a little more clear by introducing one piece of notation that will come in handy for the rest of the chapter.

Definition. Let `func` be an algorithm. For every $n \in \mathbb{N}$, we define the set $\mathcal{I}_{func,n}$ to be the set of allowed inputs to `func` of size n.

For example, $\mathcal{I}_{print\_items,100}$ is simply the set of all lists of length 100. $\mathcal{I}_{print\_items,0}$ is the set containing just one input: the empty list.

We can restate our observation about `print_items` in terms of these sets: for all $n \in \mathbb{N}$, every element of $\mathcal{I}_{print\_items,n}$ has the *same* runtime when passed to `print_items`.

Definition. Let `func` be an algorithm whose runtime depends *only* on its input size. We define the **running time function of `func`** as $RT_{func} : \mathbb{N} \to \mathbb{R}^{\geq 0}$, where $RT_{func}(n)$ is equal to the running time of `func` when given an input of size n.

The goal of a *running time analysis* for `func` is to find a function f (typically a simple elementary function) such that $RT_{func} \in \Theta(f)$.

Our first technique for performing this runtime analysis follows four steps:

1. Identify the blocks of code which can be counted as a single basic operation, because they don't depend on the input size.
2. Identify any loops in the code, which cause basic operations to repeat. You'll need to figure out how many times those loops run, based on the size of the input. Be *exact* when counting loop iterations.
3. Use your observations from the previous two steps to come up with an expression for the number of basic operations used in this algorithm—i.e., find an exact expression for $RT_{func}(n)$.
4. Use the properties of asymptotic notation to find an elementary function f such that $RT_{func} \in \Theta(f(n))$.

Because Theta expressions depend only on the fastest-growing term in a sum, *and* ignores constants, we don't even need an exact, "correct" expression for the number of basic operations. This allows us to be rough with our analysis, but still get the correct Theta expression.

Example. Consider the function `print_items`. We define input size to be the *number of items of the input list*. Perform a runtime analysis of `print_items`.

```python
def print_items(lst: list) -> None:
    for item in lst:
        print(item)
```

Running time analysis. Let n be the length of the input list `lst`.

For this algorithm, each iteration of the loop can be counted as a single operation, because nothing in it (including the call to `print`) depends on the size of the input list.[3]

[3] This is actually a little subtle. If we consider the size of individual list elements, it could be the case that some take a much longer time to print than others (imagine printing a string of one-thousand characters vs. the number 5). But by defining input size purely as the number of items, we are implicitly ignoring the size of the individual items. The running time of a call to `print` does *not* depend on the length of the input list.

So the running time depends on the number of loop iterations. Since this is a for loop over the `lst` argument.

Thus the total number of basic operations performed is n, and so the running time is $RT_{print\_items}(n) = n$, which is $\Theta(n)$. ■

Here is a second example, which has a similar structure to our first example, but also features slightly more code, using the familiar loop accumulator pattern.

Example. Analyse the running time of the following function.

```python
def my_sum(numbers: list[int]) -> int:
    sum_so_far = 0
    for number in numbers:
        sum_so_far = sum_so_far + number

    return sum_so_far
```

Running time analysis. Let n be the length of the input list (i.e., `numbers`).

This function body consists of three statements (with the middle statement, the for loop, itself containing more statements). To analyse the total running time of the function, we need to count each statement separately:

- The assignment statement `sum_so_far = 0` counts as 1 step, as its running time does not depend on the length of `numbers`.
- The for loop takes n steps: it has n iterations, and each iteration takes 1 step.[4]

[4] Remember that we're treating all arithmetic operations as constant time here.

- The return statement counts as 1 step: it, too, has running time that does not depend on the length of `numbers`.

The total running time is the sum of these three parts: $1 + n + 1 = n + 2$, which is $\Theta(n)$. ■

Nested loops

It is quite possible to have nested loops in a function body, and analyze the running time in the same fashion. The simplest method of tackling such functions is to count the number of repeated basic operations in a loop starting with the *innermost* loop and working your way out.

Example. Consider the following function.

```
def print_sums(lst: list) -> None:
    for item1 in lst:
        for item2 in lst:
            print(item1 + item2)
```

Perform a runtime analysis of `print_sums`.

Running time analysis. Let n be the length of `lst`.

The inner loop (`for item2 in lst`) runs n times (once per item in `lst`), and each iteration is just a single basic operation.

But the entire inner loop is itself repeated, since it is inside another loop. The outer loop runs n times as well, and each of its iterations takes n operations.

So then the total number of basic operations is

$$\begin{aligned} RT_{print\_sums}(n) &= \text{steps for the inner loop} \times \text{number of times inner loop is repeated} \\ &= n \times n \\ &= n^2 \end{aligned}$$

So the running time of this algorithm is $\Theta(n^2)$. ■

Students often make the mistake, however, that the number of nested loops should always be the exponent of n in the Big-O expression.[5] However, things are not that simple, and in

[5] E.g., two levels of nested loops always becomes $\Theta(n^2)$.

particular, not every loop takes n iterations.

Example. Consider the following function:

```python
def f(lst: list[int]) -> None:
    for item in lst:
        for i in range(0, 10):
            print(item + i)
```

Perform a runtime analysis of this function.

Running time analysis. Let n be the length of the input list `lst`. The inner loop repeats 10 times, and each iteration is again a single basic operation, for a total of 10 basic operations. The outer loop repeats n times, and each iteration takes 10 steps, for a total of $10n$ steps. So the running time of this function is $\Theta(n)$. (Even though it has a nested loop!)

Alternative, more concise analysis. The inner loop's running time doesn't depend on the number of items in the input list, so we can count it as a single basic operation.

The outer loop runs n times, and each iteration takes 1 step, for a total of n steps, which is $\Theta(n)$. ■

When we are analyzing the running time of two blocks of code executed in sequence (one after the other), we add together their individual running times. The sum theorems are particularly helpful here, as it tells us that we can simply compute Theta expressions for the blocks individually, and then combine them just by taking the fastest-growing one. Because Theta expressions are a simplification of exact mathematical function expressions, taking this approach is often easier and faster than trying to count an exact number steps for the entire function.[6]

[6] E.g., $\Theta(n^2)$ is simpler than $10n^2 + 0.001n + 165$.

Example. Analyze the running time of the following function, which is a combination of two previous functions.

```
def combined(lst: list[int]) -> None:
    # Loop 1
    for item in lst:
        for i in range(10):
            print(item + i)

    # Loop 2
    for item1 in lst:
        for item2 in lst:
            print(item1 + item2)
```

Running time analysis. Let n be the length of `lst`. We have already seen that the first loop runs in time $\Theta(n)$, while the second loop runs in time $\Theta(n^2)$.[7]

[7] By "runs in time $\Theta(n)$," we mean that the number of basic operations of the second loop is a function $f(n) \in \Theta(n)$.

By the *Sum of Functions* theorem from the previous section, we can conclude that `combined` runs in time $\Theta(n^2)$. (Since $n \in \mathcal{O}(n^2)$.) ■

Loop iterations with changing costs

Now let's look at one last example in this section, which is a function that prints out the sum of all distinct pairs of integers from a given list.

Example. Analyze the running time of the following function

```
def all_pairs(lst: list[int]) -> None:
    for i in range(0, len(lst)):
        for j in range(0, i):
            print(lst[i] + lst[j])
```

Discussion. Like previous examples, this function has a nested loop. However, unlike those examples, here the inner loop's running time depends on the current value of `i`, i.e., which iteration of the outer loop we're on.

This means we cannot take the previous approach of calculating the cost of the inner loop, and multiplying it by the number of iterations of the outer loop; this only works if the cost of each outer loop iteration is the same.

So instead, we need to manually add up the cost of each iteration of the outer loop, which depends on the number of iterations of the inner loop. More specifically, since j goes from 0 to $i - 1$, the number of iterations of the inner loop is i, and each iteration of the inner loop counts as one basic operation.

Let's see how to do this in a formal analysis.

Running time analysis. Let n be the length of the input list.

We start by analysing the running time of the inner loop for a *fixed* iteration of the outer loop, and a fixed value of i.

- The inner loop iterates i times (for j going from 0 to $i - 1$), and each iteration takes one step (constant time).[8] Therefore the cost of

[8] Here, list indexing is counted as constant time—we'll explore this more a bit later this chapter.

the inner loop is i steps, for one iteration of the outer loop.

Now, the outer loop iterates n times for i going from 0 to $n - 1$. But here the cost of each iteration is not constant. Instead, the cost of iteration i is i steps, and so the total cost of the outer loop is:

$$\sum_{i=0}^{n-1} i = \frac{n(n-1)}{2}$$

Here we used the summation formula for the sum of the first n natural numbers, which is reviewed in Appendix C.1.

And so the total number of steps taken by `all_pairs` is $\frac{n(n-1)}{2}$, which is $\Theta(n^2)$.[9]

[9] Note that we can write $\frac{n(n-1)}{2} = \frac{1}{2}n^2 - \frac{1}{2}n$.